Python
Lecture 3

Python Lists

Chapter 1

Introduction

What is a List?

In programming, it is common to want to work with collections of data. In Python, a *list* is one of the many built-in data structures that allows us to work with a collection of data in sequential order.

Suppose we want to make a list of the heights of students in a class:

- Noelle is 61 inches tall
- Ava is 70 inches tall
- Sam is 67 inches tall
- Mia is 64 inches tall

In Python, we can create a variable called `heights` to store these integers into a *list*:

```
heights = [61, 70, 67, 64]
```

Notice that:

1. A list begins and ends with square brackets (`[` and `]`).
2. Each item (i.e., `67` or `70`) is separated by a comma (`,`)
3. It's considered good practice to insert a space () after each comma, but your code will run just fine if you forget the space.

Let's write our own list!

Example

```
heights = [61, 70, 67, 64, 65]


broken_heights = [65, 71, 59, 62]
```

What can a List contain?

Lists can contain more than just numbers.

Let's revisit our classroom example with heights:

- Noelle is 61 inches tall
- Ava is 70 inches tall
- Sam is 67 inches tall

- Mia is 64 inches tall

Instead of storing each student's height, we can make a list that contains their names:

```
names = ["Noelle", "Ava", "Sam", "Mia"]
```

We can even combine multiple data types in one list. For example, this list contains both a string and an integer:

```
mixed_list_string_number = ["Noelle", 61]
```

Lists can contain any data type in Python! For example, this list contains a string, integer, boolean, and float.

```
mixed_list_common = ["Mia", 27, False, 0.5]
```

Let's experiment with different data types in our own lists!

Example

```
ints_and_strings = [1, 2, 3, "four", "five", "six"]


sam_height_and_testscore = ["Sam", 67, 85.5, True]
```

Empty Lists

A list doesn't have to contain anything. You can create an empty list like this:

```
empty_list = []
```

Why would we create an empty list?

Usually, it's because we're planning on filling it up later based on some other input. We'll talk about two ways of filling up a list in the next exercise.

List Methods

As we start exploring lists further in the next exercises, we will encounter the concept of a *method*.

In Python, for any specific data-type (strings, booleans, lists, etc.) there is built-in functionality that we can use to create, manipulate, and even delete our data. We call this built-in functionality a method.

For lists, methods will follow the form of `list_name.method()`. Some methods will require an input value that will go between the parenthesis of the method `( )`.

An example of a popular list method is `.append()`, which allows us to add an element to the end of a list.

```
append_example = [ 'This', 'is', 'an', 'example']
append_example.append('list')

print(append_example)
```

Will output:

```
['This', 'is', 'an', 'example', 'list']
```

Example

```
example_list = [1, 2, 3, 4]

#Using Append
example_list.append(5)
# print(example_list)

#Using Remove
example_list.remove(5)
# print(example_list)
```

Growing a List: Append

We can add a single element to a list using the `.append()` Python method.

Suppose we have an empty list called `garden`:

```
garden = []
```

We can add the element `"Tomatoes"` by using the `.append()` method:

```
garden.append("Tomatoes")

print(garden)
```

Will output:

```
['Tomatoes']
```

We see that `garden` now contains `"Tomatoes"`!

When we use `.append()` on a list that already has elements, our new element is added to the **end** of the list:

```
# Create a list
garden = ["Tomatoes", "Grapes", "Cauliflower"]

# Append a new element
garden.append("Green Beans")
print(garden)
```
Will output:

```
['Tomatoes', 'Grapes', 'Cauliflower', 'Green Beans']
```
Let's use the `.append()` method to manipulate a list.

Example

```
orders = ["daisies", "periwinkle"]
orders.append("tulips")
orders.append("roses")
print(orders)
```

Growing a List: Plus (+)
When we want to add multiple items to a list, we can use `+` to combine two lists (this is also known as concatenation).

Below, we have a list of items sold at a bakery called `items_sold`:

```
items_sold = ["cake", "cookie", "bread"]
```
Suppose the bakery wants to start selling `"biscuit"` and `"tart"`:

```
items_sold_new = items_sold + ["biscuit", "tart"]
print(items_sold_new)
```
Would output:

```
['cake', 'cookie', 'bread', 'biscuit', 'tart']
```
In this example, we created a new variable, `items_sold_new`, which contained both the original items sold, and the new items. We can inspect the original `items_sold` and see that it did not change:

```
print(items_sold)
```
Would output:

```
['cake', 'cookie', 'bread']
```
We can only use `+` with other lists. If we type in this code:

```
my_list = [1, 2, 3]
my_list + 4
```

we will get the following error:

```
TypeError: can only concatenate list (not "int") to list
```

If we want to add a single element using `+`, we have to put it into a list with brackets (`[]`):

```
my_list + [4]
```

Let's use `+` to practice combining two lists!

Example

```
orders = ["daisy", "buttercup", "snapdragon", "gardenia", "lily"]

# Create new orders here:
new_orders = ["lilac", "iris"]
orders_combined = orders + new_orders


broken_prices = [5, 3, 4, 5, 4] + [4]
```

Accessing List Elements

We are interviewing candidates for a job. We will call each candidate in order, represented by a Python list:

```
calls = ["Juan", "Zofia", "Amare", "Ezio", "Ananya"]
```

First, we'll call `"Juan"`, then `"Zofia"`, etc.

In Python, we call the location of an element in a list its *index*.

Python lists are *zero-indexed*. This means that the first element in a list has index `0`, rather than `1`.

Here are the index numbers for the list `calls`:

Element	Index
`"Juan"`	`0`
`"Zofia"`	`1`
`"Amare"`	`2`
`"Ezio"`	`3`
`"Ananya"`	`4`

In this example, the element with *index* `2` is `"Amare"`.

We can select a single element from a list by using square brackets (`[]`) and the index of the list item. If we wanted to select the third element from the list, we'd use `calls[2]`:

```
print(calls[2])
```

Will output:

```
Amare
```

Note: When accessing elements of an list, you *must* use an `int` as the index. If you use a `float`, you will get an error. This can be especially tricky when using division. For example `print(calls[4/2])` will result in an error, because `4/2` gets evaluated to the `float` `2.0`.

To solve this problem, you can force the result of your division to be an `int` by using the `int()` function. `int()` takes a number and cuts off the decimal point. For example, `int(5.9)` and `int(5.0)` will both become `5`.
Therefore, `calls[int(4/2)]` will result in the same value as `calls[2]`, whereas `calls[4/2]` will result in an error.

Example

```
employees = ["Michael", "Dwight", "Jim", "Pam", "Ryan", "Andy", "Robert"]

employee_four = employees[3]
print(employee_four)
```

Accessing List Elements: Negative Index
What if we want to select the last element of a list?

We can use the index `-1` to select the last item of a list, even when we don't know how many elements are in a list.

Consider the following list with 6 elements:

```
pancake_recipe = ["eggs", "flour", "butter", "milk", "sugar", "love"]
```

If we select the `-1` index, we get the final element, `"love"`.

```
print(pancake_recipe[-1])
```

Would output:

```
love
```

This is equivalent to selecting the element with index `5`:

```
print(pancake_recipe[5])
```

Would output:

```
love
```

Here are the negative index numbers for our list:

Element	Index
`"eggs"`	`-6`
`"flour"`	`-5`
`"butter"`	`-4`
`"milk"`	`-3`
`"sugar"`	`-2`
`"love"`	`-1`

Example

```
shopping_list = ["eggs", "butter", "milk", "cucumbers", "juice", "cereal"]

last_element = shopping_list[-1]
index5_element = shopping_list[5]
print(last_element)
```

```
print(index5_element)
```

Modifying List Elements
Let's return to our garden.

```
garden = ["Tomatoes", "Green Beans", "Cauliflower", "Grapes"]
```
Unfortunately, we forgot to water our cauliflower and we don't think it is going to recover.

Thankfully our friend Jiho from Petal Power came to the rescue. Jiho gifted us some strawberry seeds. We will replace the cauliflower with our new seeds.

We will need to modify the list to accommodate the change to our `garden` list. To change a value in a list, reassign the value using the specific index.

```
garden[2] = "Strawberries"

print(garden)
```
Will output:

```
["Tomatoes", "Green Beans", "Strawberries", "Grapes"]
```
Negative indices will work as well.

```
garden[-1] = "Raspberries"

print(garden)
```
Will output:

```
["Tomatoes", "Green Beans", "Strawberries", "Raspberries"]
```

Example

```
garden_waitlist = ["Jiho", "Adam", "Sonny", "Alisha"]
garden_waitlist[1] = "Calla"
garden_waitlist[-1] = "Alex"
print(garden_waitlist)
```

Shrinking a List: Remove
We can remove elements in a list using the `.remove()` Python method.

Suppose we have a filled list called `shopping_line` that represents a line at a grocery store:

```
shopping_line = ["Cole", "Kip", "Chris", "Sylvana"]
```
We could remove `"Chris"` by using the `.remove()` method:

```
shopping_line.remove("Chris")

print(shopping_line)
```
If we examine `shopping_line`, we can see that it now doesn't contain `"Chris"`:

```
["Cole", "Kip", "Sylvana"]
```
We can also use `.remove()` on a list that has duplicate elements.

Only the first instance of the matching element is removed:

```
# Create a list
shopping_line = ["Cole", "Kip", "Chris", "Sylvana", "Chris"]

# Remove a element
shopping_line.remove("Chris")
print(shopping_line)
```
Will output:

```
["Cole", "Kip", "Sylvana", "Chris"]
```
Let's practice using the `.remove()` method to remove elements from a list.

Example

```
order_list = ["Celery", "Orange Juice", "Orange", "Flatbread"]
order_list.remove("Flatbread")
print(order_list)
new_store_order_list = ["Orange", "Apple", "Mango", "Broccoli", "Mango"]
new_store_order_list.remove("Mango")
new_store_order_list.remove("Onions")
print(new_store_order_list)
```

Two-Dimensional (2D) Lists

We've seen that the items in a list can be numbers or strings. Lists can contain other lists! We will commonly refer to these as *two-dimensional (2D)* lists.

Once more, let's look at a class height example:

- Noelle is 61 inches tall
- Ava is 70 inches tall
- Sam is 67 inches tall
- Mia is 64 inches tall

Previously, we saw that we could create a list representing both Noelle's name and height:

```
noelle = ["Noelle", 61]
```

We can put several of these lists into one list, such that each entry in the list represents a student and their height:

```
heights = [["Noelle", 61], ["Ava", 70], ["Sam", 67], ["Mia", 64]]
```

We will often find that a two-dimensional list is a very good structure for representing grids such as games like tic-tac-toe.

```
#A 2d list with three lists in each of the indices.
tic_tac_toe = [
            [["X"],["O"],["X"]],
            [["O"],["X"],["O"]],
            [["O"],["O"],["X"]]
]
```

Let's practice creating our own 2D list!

Example

```
heights = [["Jenny", 61], ["Alexus", 70], ["Sam", 67], ["Grace", 64]]
heights.append(["Vik", 68])

ages = [["Aaron", 15], ["Dhruti", 16]]
```

Accessing 2D Lists
Let's return to our classroom heights example:

```
heights = [["Noelle", 61], ["Ali", 70], ["Sam", 67]]
```

Two-dimensional lists can be accessed similar to their one-dimensional counterpart. Instead of providing a single pair of brackets `[ ]` we will use an additional set for each dimension past the first.

If we wanted to access `"Noelle"`'s height:

```
#Access the sublist at index 0, and then access the 1st index of
that sublist.
noelles_height = heights[0][1]
print(noelles_height)
```

Would output:

```
61
```

Here are the index numbers to access data for the list `heights`:

Element	Index
`"Noelle"`	`heights[0][0]`
`61`	`heights[0][1]`
`"Ali"`	`heights[1][0]`
`70`	`heights[1][1]`
`"Sam"`	`heights[2][0]`
`67`	`heights[2][1]`

Example

```
class_name_test = [
  ["Jenny", 90],
  ["Alexus", 85.5],
  ["Sam", 83],
  ["Ellie", 101.5]
]
print(class_name_test)
```

```
sams_score = class_name_test[2][1]
print(sams_score)
ellies_score = class_name_test[-1][-1]
print(ellies_score)
```

Modifying 2D Lists

Now that we know how to access two-dimensional lists, modifying the elements should come naturally.

Let's return to a classroom example, but now instead of heights or test scores, our list stores the student's favorite hobby!

```
class_name_hobbies = [["Jenny", "Breakdancing"], ["Alexus", "Photography"], ["Grace", "Soccer"]]
```

`"Jenny"` changed their mind and is now more interested in `"Meditation"`.

We will need to modify the list to accommodate the change to our `class_name_hobbies` list. To change a value in a two-dimensional list, reassign the value using the specific index.

```
# The list of Jenny is at index 0. The hobby is at index 1.
class_name_hobbies[0][1] = "Meditation"
print(class_name_hobbies)
```

Would output:

```
[["Jenny", "Meditation"], ["Alexus", "Photography"], ["Grace", "Soccer"]]
```

Negative indices will work as well.

```
# The list of Jenny is at index 0. The hobby is at index 1.
class_name_hobbies[-1][-1] = "Football"
print(class_name_hobbies)
```

Would output:

```
[["Jenny", "Meditation"], ["Alexus", "Photography"], ["Grace", "Football"]]
```

Example

```
incoming_class = [
  ["Kenny", "American", 9],
```

```
  ["Tanya", "Russian", 9],
  ["Madison", "Indian", 7]
]
incoming_class[2][2] = 8
incoming_class[-3][-3] = "Ken"
print(incoming_class)
```

Chapter 2

Working with Lists

Now that we know how to create and access list data, we can start to explore additional ways of working with lists.

In this lesson, you'll learn how to:

- Add and remove items from a list using a specific index.
- Create lists with continuous values.
- Get the length of a list.
- Select portions of a list (called *slicing*).
- Count the number of times that an element appears in a list.
- Sort a list of items.

Note: In some of the exercises, we will be using built-in functions in Python. If you haven't yet explored the concept of a function, it may look a bit new. Below we compare it to the method syntax we learned in the earlier lesson.

Here is a preview:

```
# Example syntax for methods
list.method(input)

# Example syntax for a built-in function
builtinfuncion(input)
```

Python Lists Methods

.count() - A list method to count the number of occurrences of an element in a list.

.insert() - A list method to insert an element into a specific index of a list.

.pop() - A list method to remove an element from a specific index or from the end of a list.

range() - A built-in Python function to create a sequence of integers.

len() - A built-in Python function to get the length of a list.

.sort() / sorted() - A method and a built-in function to sort a list.

Adding by Index: Insert
The Python list method `.insert()` allows us to add an element to a specific index in a list.

The `.insert()` method takes in two inputs:

1. The index you want to insert into.
2. The element you want to insert at the specified index.

The `.insert()` method will handle shifting over elements and can be used with negative indices.

To see it in action let's imagine we have a list representing a line at a store:

```
store_line = ["Karla", "Maxium", "Martim", "Isabella"]
```

`"Maxium"` saved a spot for his friend `"Vikor"` and we need to adjust the list to add him into the line right behind `"Maxium"`.

For this example, we can assume that `"Karla"` is the front of the line and the rest of the elements are behind her.

Here is how we would use the `.insert()` method to insert `"Vikor"` :

```
store_line.insert(2, "Vikor")
print(store_line)
```

Would output:

```
 ['Karla', 'Maxium', 'Vikor', 'Martim', 'Isabella']
```

Some important things to note:

1. The order and number of the inputs is important. The `.insert()` method expects two inputs, the first being a numerical index, followed by any value as the second input.
2. When we insert an element into a list, all elements from the specified index and up to the last index are shifted one index to the right. This does not apply to inserting an element to the very end of a list as it will simply add an additional index and no other elements will need to shift.

Let's practice using `.insert()`!

Example

```
front_display_list = ["Mango", "Filet Mignon", "Chocolate Milk"]


# Your code below:
front_display_list.insert(0, "Pineapple")
print(front_display_list)
```

Removing by Index: Pop

Just as we learned to insert elements at specific indices, Python gives us a method to remove elements at a specific index using a method called `.pop()`.

The `.pop()` method takes an optional single input:

1. The index for the element you want to remove.

To see it in action, let's consider a list called `cs_topics` that stores a collection of topics one might study in a computer science program.

```
cs_topics = ["Python", "Data Structures", "Balloon Making",
"Algorithms", "Clowns 101"]
```

Two of these topics don't look like they belong, let's see how we remove them using `.pop()`.

First let's remove `"Clowns 101"`:

```
removed_element = cs_topics.pop()
print(cs_topics)
print(removed_element)
```

Would output:

```
['Python', 'Data Structures', 'Balloon Making', 'Algorithms']
'Clowns 101'
```

Notice two things about this example:

1. The method can be called without a specific index. Using `.pop()` without an index will remove whatever the last element of the list is. In our case `"Clowns 101"` gets removed.
2. `.pop()` is unique in that it will *return* the value that was removed. If we wanted to know what element was deleted, simply assign a variable to the call of the `.pop()` method. In this case, we assigned it to `removed_element`.

Lastly let's remove `"Balloon Making"`:

```
cs_topics.pop(2)
print(cs_topics)
```
Would output:

```
['Python', 'Data Structures', 'Algorithms']
```
Notice two things about this example:

1. The method can be called with an optional specific index to remove. In our case, the index `2` removes the value of `"Balloon Making"`.
2. We don't have to save the removed value to any variable if we don't care to use it later.

Note: Passing in an index that does not exist or calling `.pop()` on an empty list will both result in an `IndexError`.

Let's apply what we learned about the `.pop()` method.

Example

```
data_science_topics = ["Machine Learning", "SQL", "Pandas", "Algorithms", "Sta
tistics", "Python 3"]
print(data_science_topics)

# Your code below:
removed_topic = data_science_topics.pop()
print(removed_topic)

data_science_topics.pop(3)
print(data_science_topics)
```

Consecutive Lists: Range

Often, we want to create a list of consecutive numbers in our programs. For example, suppose we want a list containing the numbers 0 through 9:

```
my_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```
Typing out all of those numbers takes time and the more numbers we type, the more likely it is that we have a typo that can cause an error.

Python gives us an easy way of creating these types of lists using a built-in function called `range()`.

The function `range()` takes a single input, and generates numbers starting at `0` and ending at the number **before** the input.

So, if we want the numbers from `0` through `9`, we use `range(10)` because `10` is 1 greater than `9`:

```
my_range = range(10)
print(my_range)
```

Would output:

```
range(0, 10)
```

Notice something different? The `range()` function is unique in that it creates a *range object*. It is not a typical list like the ones we have been working with.

In order to use this object as a list, we have to first convert it using another built-in function called `list()`.

The `list()` function takes in a single input for the object you want to convert.

We use the `list()` function on our range object like this:

```
print(list(my_range))
```

Would output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Let's try out using `range()`!

Example

```
number_list = range(0, 9)
print(list(number_list))
zero_to_seven = range(0, 8)
print(list(zero_to_seven))
```

The Power of Range!

By default, `range()` creates a list starting at `0`. However, if we call `range()` with two inputs, we can create a list that starts at a different number.

For example, `range(2, 9)` would generate numbers starting at `2` and ending at `8` (just before `9`):

```
my_list = range(2, 9)
print(list(my_list))
```

Would output:

```
[2, 3, 4, 5, 6, 7, 8]
```

If we use a third input, we can create a list that "skips" numbers.

For example, `range(2, 9, 2)` will give us a list where each number is `2` greater than the previous number:

```
my_range2 = range(2, 9, 2)
print(list(my_range2))
```

Would output:

```
[2, 4, 6, 8]
```

We can skip as many numbers as we want!

For example, we'll start at `1` and skip in increments of `10` between each number until we get to `100`:

```
my_range3 = range(1, 100, 10)
print(list(my_range3))
```

Would output:

```
[1, 11, 21, 31, 41, 51, 61, 71, 81, 91]
```

Our list stops at `91` because the next number in the sequence would be `101`, which is greater than `100` (our stopping point).

Let's experiment with our additional `range()` inputs!

Example

```
range_five_three = range(5, 15, 3)
range_diff_five = range(0, 40, 5)
```

Length
Often, we'll need to find the number of items in a list, usually called its *length*.

We can do this using a built-in function called `len()`.

When we apply `len()` to a list, we get the number of elements in that list:

```
my_list = [1, 2, 3, 4, 5]

print(len(my_list))
```
Would output:

```
5
```
Let's find the length of various lists!

Example

```
long_list = [1, 5, 6, 7, -
23, 69.5, True, "very", "long", "list", "that", "keeps", "going.", "Let's", "p
ractice", "getting", "the", "length"]

range_list = range(2, 3000, 100)

# Your code below:
long_list_len = len(long_list)
print(long_list_len)
range_list_length = len(range_list)
print(range_list_length)
```

Slicing Lists I
In Python, often we want to extract only a portion of a list. Dividing a list in such a manner is referred to as *slicing*.

Lets assume we have a list of `letters`:

```
letters = ["a", "b", "c", "d", "e", "f", "g"]
```
Suppose we want to select from `"b"` through `"f"`.

We can do this using the following syntax: `letters[start:end]`, where:

- `start` is the index of the first element that we want to include in our selection. In this case, we want to start at `"b"`, which has index `1`.

- `end` is the index of *one more than* the last index that we want to include. The last element we want is `"f"`, which has index `5`, so `end` needs to be `6`.

```
sliced_list = letters[1:6]
print(sliced_list)
```
Would output:

```
["b", "c", "d", "e", "f"]
```
Notice that the element at index `6` (which is `"g"`) is *not* included in our selection.

Example

```
suitcase = ["shirt", "shirt", "pants", "pants", "pajamas", "books"]

beginning = suitcase[0:2]

# Your code below:
print(beginning)

middle = ["pants", "pants"]
print(middle)
```

Slicing Lists II
Slicing syntax in Python is very flexible. Let's look at a few more problems we can tackle with slicing.

Take the list `fruits` as our example:

```
fruits = ["apple", "cherry", "pineapple", "orange", "mango"]
```
If we want to select the *first `n` elements* of a list, we could use the following code:

```
fruits[:n]
```
For our `fruits` list, suppose we wanted to slice the first three elements.

The following code would start slicing from index `0` and up to index `3`. Note that the fruit at index `3` (`orange`) is not included in the results.

```
print(fruits[:3])
```
Would output:

```
['apple', 'cherry', 'pineapple']
```
We can do something similar when we want to slice the *last `n` elements* in a list:

```
fruits[-n:]
```
For our `fruits` list, suppose we wanted to slice the last two elements.

This code slices from the element at index `-2` up through the last index.

```
print(fruits[-2:])
```
Would output:

```
['orange', 'mango']
```
Negative indices can also accomplish taking *all but n last elements* of a list.

```
fruits[:-n]
```
For our `fruits` example, suppose we wanted to slice all but the last element from the list.

This example starts counting from the `0` index up to the element at index `-1`.

```
print(fruits[:-1])
```
Would output:

```
['apple', 'cherry', 'pineapple', 'orange']
```
Let's practice some of these extra slicing techniques!

Example

```
suitcase = ["shirt", "shirt", "pants", "pants", "pajamas", "books"]

# Your code below:
last_two_elements = suitcase[-2:]
print(last_two_elements)

slice_off_last_three = suitcase[:-3]
print(slice_off_last_three)
```

Counting in a List

In Python, it is common to want to count occurrences of an item in a list.

Suppose we have a list called `letters` that represents the letters in the word "Mississippi":

```
letters = ["m", "i", "s", "s", "i", "s", "s", "i", "p", "p", "i"]
```

If we want to know how many times `i` appears in this word, we can use the list method called `.count()`:

```
num_i = letters.count("i")
print(num_i)
```

Would output:

```
4
```

Notice that since `.count()` *returns* a value, we can assign it to a variable to use it.

We can even use `.count()` to count element appearances in a two-dimensional list.

Let's use the list `number_collection` as an example:

```
number_collection = [[100, 200], [100, 200], [475, 29], [34, 34]]
```

If we wanted to know how often the sublist `[100, 200]` appears:

```
num_pairs = number_collection.count([100, 200])
print(num_pairs)
```

Would output:

```
2
```

Let's count some list items using the `.count()` method!

Example

```
votes = ["Jake", "Jake", "Laurie", "Laurie", "Laurie", "Jake", "Jake", "Jake",
 "Laurie", "Cassie", "Cassie", "Jake", "Jake", "Cassie", "Laurie", "Cassie", "
Jake", "Jake", "Cassie", "Laurie"]

jake_votes = votes.count("Jake")
print(jake_votes)
```

Sorting Lists I

Often, we will want to sort a list in either numerical (1, 2, 3, ...) or alphabetical (a, b, c, ...) order.

We can sort a list using the method `.sort()`.

Suppose that we have a list of names:

```
names = ["Xander", "Buffy", "Angel", "Willow", "Giles"]
```

Let's see what happens when we apply `.sort()`:

```
names.sort()
print(names)
```

Would output:

```
['Angel', 'Buffy', 'Giles', 'Willow', 'Xander']
```

As we can see, the `.sort()` method sorted our list of `names` in alphabetical order.

`.sort()` also provides us the option to go in reverse. Instead of sorting in ascending order like we just saw, we can do so in descending order.

```
names.sort(reverse=True)
print(names)
```

Would output:

```
['Xander', 'Willow', 'Giles', 'Buffy', 'Angel']
```

Note: The `.sort()` method does not return any value and thus does not need to be assigned to a variable since it modifies the list directly. If we do assign the result of the method, it would assign the value of `None` to the variable.

Let's experiment sorting various lists!

Example

```
# Checkpoint 1 & 2
addresses = ["221 B Baker St.", "42 Wallaby Way", "12 Grimmauld Place", "742 E
vergreen Terrace", "1600 Pennsylvania Ave", "10 Downing St."]
addresses.sort()
print(addresses)


# Checkpoint 3
names = ["Ron", "Hermione", "Harry", "Albus", "Sirius"]
names.sort()
```

```
# Checkpoint 4 & 5
cities = ["London", "Paris", "Rome", "Los Angeles", "New York"]
cities.sort(reverse=True)
print(cities)
```

Sorting Lists II

A second way of sorting a list in Python is to use the built-in function `sorted()`.

The `sorted()` function is different from the `.sort()` method in two ways:

1. It comes *before* a list, instead of after as all built-in functions do.
2. It generates a new list rather than modifying the one that already exists.

Let's return to our list of names:

```
names = ["Xander", "Buffy", "Angel", "Willow", "Giles"]
```

Using `sorted()`, we can create a new list, called `sorted_names`:

```
sorted_names = sorted(names)
print(sorted_names)
```

This yields the list sorted alphabetically:

```
['Angel', 'Buffy', 'Giles', 'Willow', 'Xander']
```

Note that using `sorted` did not change `names`:

```
print(names)
```

Would output:

```
['Xander', 'Buffy', 'Angel', 'Willow', 'Giles']
```

Example

```
games = ["Portal", "Minecraft", "Pacman", "Tetris", "The Sims", "Pokemon"]

games_sorted = sorted(games)
print(games)
print(games_sorted)
```

Lecture 3 Project 1

Gradebook

You are a student and you are trying to organize your subjects and grades using Python. Let's explore what we've learned about lists to organize your subjects and scores.

Create Some Lists:

1 Create a list called `subjects` and fill it with the classes you are taking:

- `"physics"`
- `"calculus"`
- `"poetry"`
- `"history"`

2 Create a list called `grades` and fill it with your scores:

- `98`
- `97`
- `85`
- `88`

3 Manually (without any methods) create a two-dimensional list to combine `subjects` and `grades`. Use the table below as a reference to associated values.

Name	Test Score
`"physics"`	`98`
`"calculus"`	`97`
`"poetry"`	`85`
`"history"`	`88`

Assign the value into a variable called `gradebook`.

4 Print `gradebook`.
Does it look how you expected it would?

Add More Subjects:

5 Your grade for your computer science class just came in! You got a perfect score, `100`!
Use the `.append()` method to add a list with the values of `"computer science"` and an associated grade value of `100` to our two-dimensional list of `gradebook`.

6 Your grade for `"visual arts"` just came in! You got a 93!
Use `append` to add `["visual arts", 93]` to `gradebook`.

Modify The Gradebook:

7 Our instructor just told us they made a mistake grading and are rewarding an extra 5 points for our visual arts class.
Access the index of the grade for your visual arts class and modify it to be 5 points greater.

8 You decided to switch from a numerical grade value to a Pass/Fail option for your poetry class.
Find the grade value in your `gradebook` for your poetry class and use the `.remove()` method to delete it.

9 Use the `.append()` method to then add a new `"Pass"` value to the sublist where your poetry class is located.

10 You also have your grades from last semester, stored in `last_semester_gradebook`.
Create a new variable `full_gradebook` that combines both `last_semester_gradebook` and `gradebook` using `+` to have one complete grade book.

Print `full_gradebook` to see our completed list.

Lecture 3 Project 2

Learning programming languages is not only about typing in class or at home. It also involves lot of research to improve knowledge. As an assignment, carry out research on other array methods such as *tuples* and *sets*. Find out the differences between *lists*, *tuples* and *sets*, and write small programming projects with them.